Introduction

Hyperparameters are the variables that govern the training process itself. They are tuned by running through the whole training job, looking at the aggregate accuracy, and adjusting with the goal of modifying the composition of the model in an effort to find the best combination to handle the given problem.

In Part 1 of this project, we tune and study the impact of the hyper-parameters max tree-depth and number of principle components for a Decision-Tree classifier, especially with varying sizes of data sets. Then in Part 2, we derive and code our own multi-class based on a cross-entropy error function.

Part 1: Hyperparameter tuning of a Decision-Tree Classifier

Although high predictive accuracy is the most frequently used measure to evaluate learning algorithms, in many applications, easy interpretation of the induced models is also an important requirement. Good predictive performance and model interpretability are found in one of the most successful set of classification algorithms: Decision Tree (DT) algorithms. When applied to a dataset, these algorithms induce a model represented by a set of rules in a tree-like structure [1].

Our general methodology is that for a dataset of a specific size, compute the cross-validation (CV) score for each element in the hyper-parameter search space and select the best model based on the accuracy CV score. We also plot the train and validation curves to qualitatively understand overfitting and underfitting in a particular model and decide on changes accordingly if needed.

We use $GridSearchCV$ in $sklearn$ with the following steps:
* Manually set a grid of discrete hyperparameter values
* Set a metric for scoring model performance
* Search exhaustively through the grid
* For each set of hyperparameters, evaluate each model's CV score
* The optimal hyperparameters are those of the model achieving the best CV score

Part 1.1: Ntrain+Nval=1000, Nvalid=1000

We first train a Decision-Tree classifier with default settings.
NB: the 'Test accuracy' mentioned henceforth is from a validation set until section 'Final validation of the model' where the actual est set put aside in the beginning is used.

```
Depth of tree in base classifier:  12
CV accuracy: 0.66 (+/- 0.06)
Test accuracy:  0.669
```

Optimizing the parameter `max_depth`

An exhaustive grid search for optimum max_depth hyperparameter results in the following:

```
Best hyper-parameters found:  {'max_depth': 8}
Best CV score:  0.683
Test accuracy:  0.68
```

We note that given the small size of the dataset, overfitting is expected. In general, the deeper we allow the tree to grow, the more complex our model will become because we will have more splits and it captures more information about the data. This is one of the root causes of overfitting in decision trees.

The results of the optimal depth search are illustrated in Figure 1 and 2. Firstly, we see that the validation accuracy peaks after a max depth of 8. As mentioned, this is due to the small size of the data-set. Importantly, next we observe the gap between the training and validation accuracy. This clearly indicates that our model is overfitting on the data.

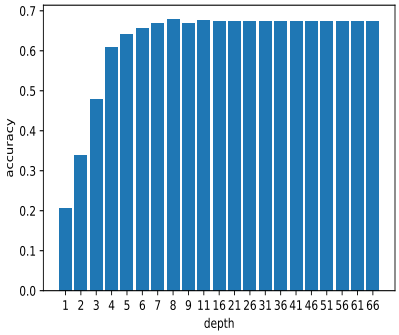


Figure 1: Max depth tuning

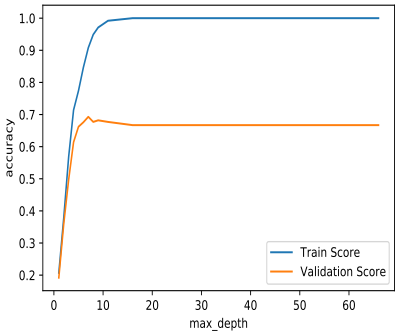


Figure 2: Train and Validation score for varying max depth

Given the constraint of the size of the dataset, we look at dimensionality reduction for aid. So next we search for the optimum number of Principle Components again using GridSearch.

Optimizing the number of Principle Components

Plotting the number of components that explain 95% of the total variance, we see that about 140 components suffice. Indeed a trait of image data.

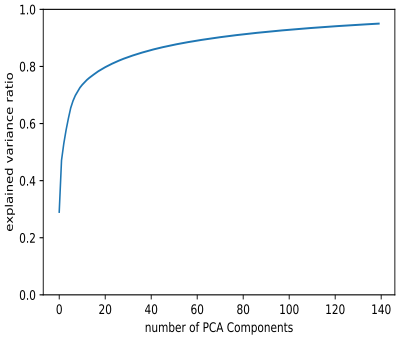


Figure 3: Variance explained by number of PCA Comp.

Fixing the max tree depth to first 5 and then 12, we search for the optimum number of principle components. The results are summarized in Table I.

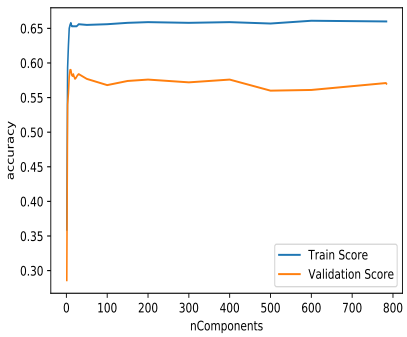
Table I

	Max Tree Depth	CV Score %	Test Score %	Optimum Principle Components
Results without PCA	12	66 (+/- 0.06)	66.7	-
	9	68.3 (+/- 0.065)	68.0	-
Results with PCA	5	60.2 (+/- 0.073)	58.8	9
	12	66.9 (+/- 0.08)	67.0	17

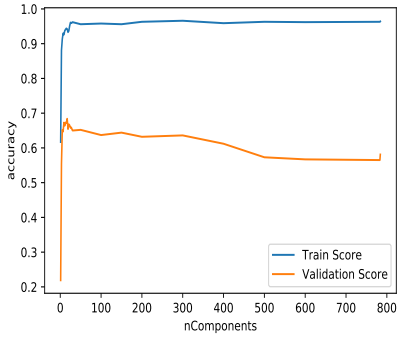
With a max depth of 5, we specify that we get up to 67% accuracy with the same depth without PCA. Here we observe a score of 53.7% for all 784 components (refer mini-project.ipynb). The discrepancy must be attributed to the rotation about the principle axes.

But more importantly, we observe that the train score is now lower and the so is the gap between train and validation score (see Figure 4(a)). Hence it can be said that shifting towards the regime of underfitting from overfitting, i.e. an increase in model power should surely improve the results.

And indeed, for a max depth of 12 (increase in model complexity), we see an improved validation accuracy of 66.9% and test score of about 67% as expected. Thus we're not far from out original score without PCA, only now we have only 17 features instead of 784. But again, from Figure 4(b), we now again observe significant overfitting.



(a) Max Depth = 5



(b) Max Depth = 12

Figure 4: Accuracy for varying no. of principle components

Part 1.2 & 1.3: Optimal number of principle components with changes in max_depth

We note that the optimal number of principle components increase from 9 to 17 for a change in tree max_depth 5 to 12. This increase is because a tree has a maximum of depth equal to the number of features of data it models, so by adding more features (i.e. increasing components) we are increasing the total depth that the tree can achieve, which in general means a more complex model.

Optimal number of principle components and max_depth

Again using grid-search on a 2D search space for `(max_depth, nComp_PCA)`, we get the following results:

```
Best hyper-parameters found:  {'clf__max_depth': 10, 'pca__n_components': 17}
Best CV score:  0.69
Test accuracy:  0.665
```

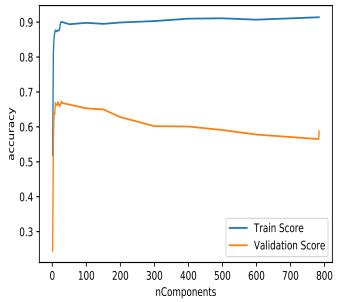
We don't see much changes from before and it is fair to conclude that the real constraint is the size of the data-set. So we next find the optimal `(max_depth, nComp_PCA)` pair for

* Ntrain+Nval=2000, Nvalid=2000
* Ntrain+Nval=20000, Nvalid=10000
* Ntrain+Nval=200, Nvalid=2000

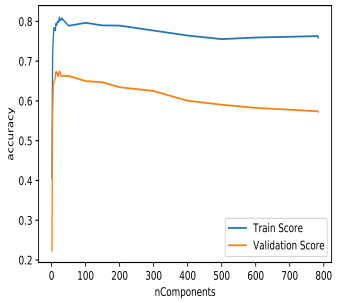
(refer to Table II and Figure 5)

Table II

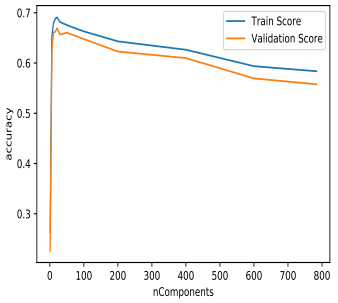
N	Max Tree Depth	CV Score %	Test Score %	Optimum Principle Components
200	10	64.0 (+/- 0.045)	57.5	7
1000	10	66.9 (+/- 0.045)	65.0	17
2000	11	71.4 (+/- 0.038)	68.5	21
20000	13	76.4 (+/- 0.01)	76.3	30



(a) N = 1000



(b) N = 2000



(c) N = 20,000

Figure 5: Accuracy for varying no. of principle components for optimal max depth of trees

As expected, the generalization capability of the model increases as it is trained on more data. With regards to the change in the hyper-parameters, we see that the total number of principle components retained is higher with a larger data-set. This is due to the larger variance of the data in the bigger sets. Also in general, the max_depth is observed to increase which can be attributed to the same reason.

This increase in generalization capability goes hand in hand with no more overfitting as can be seen from Figure 5.

Part 1.4: Final validation of the best model

We select the model with the best generalization capability, which in our case is the one trained on 20,000 examples.

```
Finally, the test accuracy for model with tuned hyper-parameters and
trained on N=1000:  0.6601
trained on N=2000:  0.6717
trained on N=20000:  0.7539
```

It is not surprising to observe this test accuracy which is not much different from the one we obtained using cross-validation. A client for this model must realize that the level of accuracy that can be achieved is about 75%.

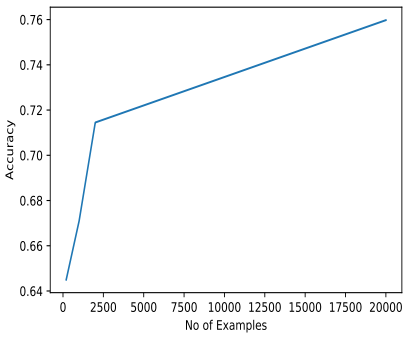


Figure 6: Effect of size of data-set on accuracy

Part 1.4 Bonus: Value vs feasibility of using more data

As can be seen from Table II, with just 200 examples, we observe a CV score of about 64% and a test score of about 57%, which is about 20% below the result with 20,000 examples. Given this, it is appealing to think if getting more data, especially when infeasible is worth it.
Atleast for the model trained on 200 examples, its result is not reliable as we can see from precision and recall of the model which are very different.

	precision	recall	f1-score	support
0	0.52	0.64	0.57	1000
1	0.79	0.76	0.77	1000
2	0.41	0.55	0.47	1000
3	0.52	0.49	0.50	1000
4	0.42	0.34	0.38	1000
5	0.71	0.54	0.61	1000
6	0.26	0.17	0.21	1000
7	0.67	0.52	0.59	1000
8	0.73	0.71	0.72	1000
9	0.58	0.91	0.71	1000
accuracy			0.56	10000
macro avg	0.56	0.56	0.55	10000
weighted avg	0.56	0.56	0.55	10000

But still, this is not the case with the model trained on 2000 examples and going from this model to the one trained on 20,000 example must be really thought through (the difference is of about 4%) taking into consideration the cost, time, and value from the minor increase in generalization accuracy.

Part 2: Multi-class Perceptron

In this part of the project, we derive a multi-class perceptron algorithm and then code it.

Part 2.1

The parameters of the model

Being a multi-class classification, let D be the dimension of the data and K the number of classes, then we have $D * K$ weight parameters $\vec{w}_k$ to be optimized. For Fashion-MNIST dataset, this implies $784 * 10$ weights to be optimized.

Derivation of the gradient's update-rule

For K $>$ 2, the soft-max function can be thought of as a generalization of sigmoid function representing a mapping from real number to a finite interval of [0, 1], like a probability of belonging to a particular class given a data point.

$$y_k^{(n)} = \text{softmax}\big((\vec{w}_k \cdot \vec{x}^{(n)})_{k=1\ldots K}\big) = \frac{\exp(\vec{w}_k \cdot \vec{x}^{(n)})}{\sum_\ell \exp(\vec{w}_\ell \cdot \vec{x}^{(n)})}$$

We can thus use cross-entropy error function to determine the parameters $\vec{w}_k$

$$E(\vec{w}_1, \vec{w}_2, ..., \vec{w}_k) = -\frac{1}{N}\sum_n^N \sum_k^K t_{n,k} \log(y_{n,k})$$

where $\vec{t}_{n,k}$, a binary vector represents a target for a feature vector x_n belonging to class C_k.
Let $a = \vec{w_k}^T x_n$, then taking the gradient of the error function, we get:

$$\nabla_{a_k} E = -\nabla_{a_k} \frac{1}{N}\sum_n^N \sum_k^K t_{n,k} \log(\frac{\exp(a_\ell)}{\sum_\ell \exp(a_\ell)})$$

where $\ell = 1, 2, ..., K$ for K classes and k is denoting a particular class instance. Consider for a moment a single class k (refer to Appendix A). Expanding the log term,

$\frac{\partial}{\partial_{a_k}} \log(\frac{\exp(a_\ell)}{\sum_\ell \exp(a_\ell)})$
$= \frac{\partial}{\partial_{a_k}} [\log(\exp(a_\ell) - \log(\sum_\ell \exp(a_\ell))]$
$= \frac{\partial}{\partial_{a_k}} (a_\ell) - \frac{\exp(a_\ell)}{\sum_\ell \exp(a_\ell)}$
$= \delta_{\ell k} - y_{\ell,n}$

where $\delta_{\ell k}$ is the Kronecker's delta function. In general,

$\nabla_{a_k} E = -\frac{1}{N} \sum_n^N \sum_k^K t_{n,k} [\delta_{\ell,k} - y_{\ell,n}]$
$= \frac{1}{N} \sum_n^N [t_{n,k} - y_{\ell,n} \sum_k^K t_{n,k}]$

Using chain rule,

$\nabla_{w_\ell} E = -\frac{1}{N} \sum_n^N \frac{\delta E}{\delta a_\ell} \cdot \frac{\delta a_\ell}{\delta w_\ell}$
$= -\frac{1}{N} \sum_n^N [t_{n,k} - y_{\ell,n} \sum_k^K t_{n,k}] x_n$

Now since $t_{n,k}$ is a binary encoded vector for each class k, $\sum_k^K t_{n,k} = 1$

Hence, the update rule for the multi-class perceptron is as follows:

$$\vec{w}_\ell \mapsto \vec{w}_\ell + \eta \frac{1}{N} \sum_n^N [t_{n,k} - y_{\ell,n}] x_n$$

Interestingly, we see that the gradient for cross-entropy has the same form as that for a mean-squared error loss function. This implies that the the function has a positive semi-definite Hessian and is convex, which guarantees a global optima and fast convergence.

Part 2.2

To code the multi-class classifier we have implemented the following methods inside the class `MultiClassPerceptron`.

Definition of the class

- `__init__(self, lr=0.03, max_iter = 200)`
  Constructor of the class
- `fit_val(self, x_train_, y_train_, x_val_, y_val_)`
  fit method which calculates the score of the algorithm on each iteration for the training and validation sets
- `fit_cross_v(self, x_train_, y_train_, n_folds=5)`
  fit method which calculates the cross validation score
- `predict(self, x)`
  method to predict the outcome y from a data-set x
- `score(self, x, y)`
  method to calculate the accuracy score of a given dataset with its ground truths.
- `_init_weights(self, d, k)`
  method to initialize the weights randomly following a standard normal distribution
- `_softmax(self, labels)`
  method to calculate the probabilities given an array of numbers following the soft-max function
- `_one_hot_enconder(self, y)`
  method to encode the variable y in a one-hot format
- `_loss_function(self, ground_truths, preds)`
  method which calculates the loss

Results

We have trained the data-set with a 5-fold cross-validation approach and a train-validation one. We can see in the figures below the accuracy scores by both of the models, the first one trained over 500 epochs and the second one over 1000.

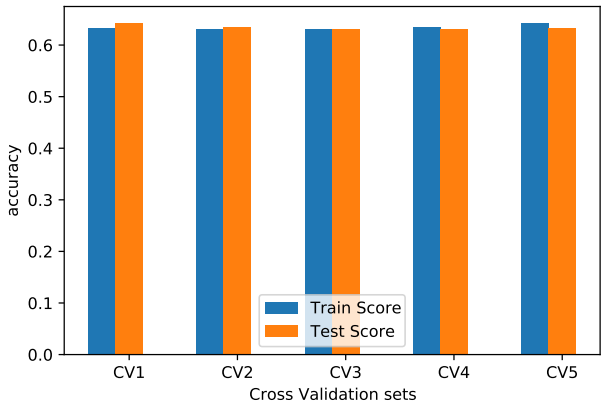


Figure 7: Cross Validation Scores in 500 epochs

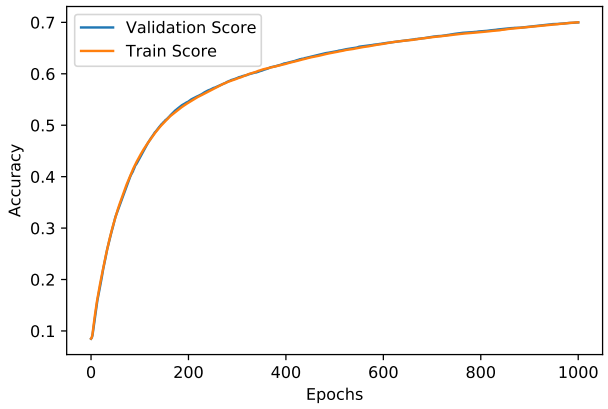


Figure 8: Train-Validation Accuracy over the Epochs

References

[1] Mantovani, R., Horváth, T., et al, 'An empirical study on hyperparameter tuning of decision trees'

Appendix A

Derivation of the update rule for the multi-class perceptron on paper. Note that same has been put more concisely in the main text for sake of brevity: